快速入门（中文）

NMRED
www.swanlinux.net

目錄

Introduction 1.1
安装 SBT 1.2
Mac 平台安装 SBT 1.2.1
Windows 平台安装 SBT 1.2.2
Linux 平台安装 SBT 1.2.3
手动安装 SBT 1.2.4
简单例子：Hello World 1.3
目录结构 1.4
运行 SBT 1.5
配置文件 .sbt 1.6
配置作用域 1.7
配置参数的方法 1.8
Lib库依赖 1.9
多项目构建 1.10
插件使用 1.11
自定义配置和任务 1.12
配置文件 .scala 1.13
总结 1.14

SBT 快速入门

本篇文件是由Getting Started summary翻译而来。

Github

https://github.com/nmred/sbt-manual

校稿

对于翻译有任何问题，可以再 Issue 中提出, 或者直接Pull requests 也可以直接微博 @虚风竹叶

Email: nmred.hao@gmail.com

版本

V0.1.1 (初稿)

介绍

SBT 是一个灵活强大的项目构建工具，相比其他构建工具用起来会发现简洁简洁，但是从功能上看一点都逊色于其他构建工具。

快速入门可以帮助您快速的利用SBT构建或维护一个工程，并且简单介绍了一些配置构建工具的相关概念。

如果你已经熟悉SBT使用，可以直接去看 配置文件 .sbt 、 配置作用域和配置参数的方法相关章节，不过建议还是按照快速入门章节顺序阅读，这样可以理解SBT的相关概念。

感谢尝试 *SBT* 并体验其中的乐趣！

安装 SBT

创建一个用 SBT 构建的工程大致需要如下几步：

- 安装 SBT 并创建一个启动脚本
- 创建一个简单的项目，以 Hello World 为例
  - 创建项目目录和项目代码相关文件
  - 配置项目构建定义文件
- 参考运行 SBT章节学习SBT如何运行
- 参考配置文件 .sbt章节学习更多的 SBT 相关定义

基本上SBT的安装可以归纳为一个 Jar 文件和一个启动脚本，但是依赖于具体的平台，我们提供了几种平台的安装步骤，在此不累赘叙述了。

Mac 平台安装 SBT

通过第三方的包安装

```
注意：第三方的包可能没有提供最新版本，可以将相关任何问题反馈给包相关的维护者
```

通过 Macports 安装

```
$ port install sbt
```

通过 Homebrew 安装

```
$ brew install sbt
```

通过通用的包安装

下载 ZIP 包或 TGZ 包解压

手动安装

参考手动安装 SBT

Windows 平台安装 SBT

通过 Windows 安装包安装

下载 msi 安装包并安装

通过通用的包安装

下载 ZIP 包或 TGZ 包解压

手动安装

参考手动安装 SBT

Linux 平台安装 SBT

通过通用的包安装

下载 ZIP 包或 TGZ 包解压

RPM 和 DEB

- RPM包
- DEB包

> 注意： 请将任何和这两个包相关的问题反馈到sbt-launcher-package项目 issue

Gentoo

In the official tree there is no ebuild for sbt. But there are ebuilds to merge sbt from binaries. To merge sbt from this ebuilds you can do:

```
$ mkdir -p /usr/local/portage && cd /usr/local/portage
$ git clone git://github.com/whiter4bbit/overlays.git
$ echo "PORTDIR_OVERLAY=$PORTDIR_OVERLAY /usr/local/portage/overlays" >> /etc/make.con
f
$ emerge sbt-bin
```

> 注意: 有任何和 ebuild 相关的问题请反馈到 ebuild issue

手动安装

参考手动安装 SBT

手动安装 SBT

Unix

将sbt-launch.jar包放到目录 ~/bin中

创建一个运行jar包的脚本 ~/bin/sbt, 脚本内容为：

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M
"
java $SBT_OPTS -jar `dirname $0`/sbt-launch.jar "$@"
```

确保脚本有执行权限

```
$ chmod u+x ~/bin/sbt
```

简单例子：Hello World

创建项目目录和项目代码

一个合法的 sbt 项目可以在一个项目目录中包含单个文件。尝试创建一个包含hw.scala 文件的目录hello, 文件中的内容如下：

```
object Hi {
    def main(args: Array[String]) = println("Hi!")
}
```

现在可以进入目录 hello 运行 sbt 命令， 在sbt交互模式下运行run 命令， 具体的在 Unix或 OS X 中的命令如下：

```
$ mkdir hello
$ cd hello
$ echo 'object Hi { def main(args: Array[String]) = println("Hi!") }' > hw.scala
$ sbt
...
> run
...
Hi!
```

在这种情况下 sbt 完全遵循一套构建规则的，sbt 会自动根据规则进行构建，具体的规则如下：

- 代码源文件可以是 sbt 项目根目录
- 代码源文件可以是 在 src/main/scala 或 src/main/java 目录
- 测试代码目录为 src/test/scala 或 src/test/java 目录
- 数据文件在 src/main/resources 或 src/test/resources
- 依赖的 jars 文件可以放到 lib 目录下

默认情况下 sbt 构建的项目用的 scala 版本和 sbt 自身运行的scala版本是一样的，可以通过运行 sbt run 命令或 sbt console 进入 Scala REPL 模式下运行项目， sbt 会加载依赖的 classpath ，所以可以使用 sbt 直接运行测试项目 。

构建项目的配置文件

许多项目都需要手动进行配置，最基本的配置一般都是定义在根目录的 build.sbt 文件中， 例如， 如果项目跟目录为 hello ， 在 hello/build.sbt 中可能为：

```
name := "hello"

version := "1.0"

scalaVersion := "2.10.3"
```

需要注意的是每个配置项之间用空行分割，这个不仅仅是为了显示，实际上 sbt 需要根据空行来分割多个配置项的。在配置文件 .sbt 章节中你可以学到如何配置 build.sbt

如果你需要将项目打包成 jar 包，需要在 build.sbt 中指定名称和最新版本号。

设置 SBT 版本

可以强制使用某个 sbt 版本在构建项目的时候，需要在 hello/project/build.properties 文件中配置：

```
sbt.version = 0.13.5
```

强制使用 sbt 的 0.13.5 版本， 虽然sbt 版本间 99% 是兼容的，不过设置 hello/project/build.properties 指定 sbt 版本可以避免版本之间不兼容导致的一些潜在问题。

目录结构

根目录

在 sbt 术语中 “根目录”是一个包含项目的目录，所以如果创建一个 hello 项目将包含 hello/build.sbt 和 hello/hw.scala 在 hello world 项目例子中，其中hello是根目录

源代码目录结构

源代码可以放到项目的根目录类似于 hello/hw.scala , 但是在眞正的项目很少利用这样的代码结构，这样会使项目变得混乱， sbt 的项目目录结构默认情况下和 Maven 一样（所有路劲是基于根目录的相对路劲）:

```
src/
    main/
        resources/
            <包含在main 的jar包中的文件>
        scala/
            <scala源代码>
        java/
            <java 源代码>
    test/
        resources/
            <包含在test 的jar包中的文件>
        scala/
            <scala 源代码>
        java/
            <java 源代码>
```

除 src/ 目录以外的目录将被忽略，包括隐藏的目录。

sbt 构建定义文件

你已经在项目的根目录中看到了 build.sbt ， 其他的 sbt 定义文件在子目录 project 中， `project` 可以包含 `.scala` 文件，将和 `.sbt` 定义进行合并来完成构建定义，详细的可以参考 .scala 配置定义

```
    build.sbt
    project/
        Build.scala
```

你可能看到在 `project/` 目录中有一个 `.sbt` 文件，这个文件和根目录中的 `.sbt` 不是针对一个项目的定义，稍后会解释这一点

构建项目

生成的文件(编译后的 class文件，jar 包，项目管理文件，缓存文件和文档)将被写入到一个 `target` 目录默认

配置项目版本控制

项目的 `.gitignore` 文件中应该包含 `target/` , 注意：以 `/` 结尾（匹配目录中所有目录和文件）并且开头不包含 `/` （为了匹配 project/target/）

运行 SBT

交互模式

在根目录中运行 `sbt` 命令不带任何参数将进入交互模式

```
$ sbt
```

交互模式有一个命令输入功能（可以用 Tab 补全和历史命令），例如，当输入 `compile` 时：

```
> compile
```

如果再次编译只需按 向上键 + 回车键 如果运行该项目输入 `run` 如果退出交互模式输入 `exit` 或用快捷键 `Ctrl+D` (Unix) 或 `Ctrl+Z` (Windows)

批量脚本模式

你也可以在批量脚本模式下运行 sbt, 指定一个用空格分割的一系列命令作为参数. 对于 sbt 命令本身也可以指定对应的参数，将命令和命令参数用双引号括起来，其中一个参数为命令，其余的为命令参数，例如：

```
$ sbt clean compile "testOnly TestA TestB"
```

在这个例子中，testOnly有两个参数分别是 TestA 和 TestB. 这个命令将按照 `clean, compile, testOnly` 顺序来执行.

持续构建于测试

为了提供 编辑-编译-测试 整个周期的效率，可以使用 sbt 的自动触发编译和运行过程当保存源代码文件的时候。使一个或多个源代码文件修改后可以自动指定对应的命令，只需在对应命令前加 `~` 前缀. 例如，在交互模式下：

```
> ~ compile
```

按回车键定制监视文件的改变

常用命令

`clean`

删除所有构建生成的文件(在target目录中)

`compile`

编译项目源代码(编译 src/main/scala 和 src/main/java 目录下的源代码)

`test`

编译并运行所有的测试用例

`console`

启动一个 Scala 语言交互模式， sbt 在启动的时候会指定依赖的所有classpath, 返回 sbt 可以用 `:quit` 、 `Ctrl+D` (Unix) 和 `Ctrl+Z` (Windows)

`run <arguments>*`

运行项目在虚拟机中

`package`

创建一个 jar 包其中包含 `src/main/resources` 和编译 `src/main/scala` 或 `src/main/java` 目录的 `class` 文件

`help <command>`

显示指定命令的帮助信息，如果没有指定命令将显示所有的命名的摘要信息。

`reload`

重新加载配置文件(build.sbt， project/.scala 和 project/.sbt 文件)， 当修改配置文件的时候需要执行

Tab 补全

在交互模式下 sbt 支持 Tab 补全功能，当按一次 Tab 键是 sbt 会显示所有可能匹配的子集命令，当按多次 Tab 后将显示多个可能匹配的命令进行选择的提示，和 Unix tab 补全规则基本一致

历史命令

在交互模式下可以 sbt 会记录历史命令，甚至是退出sbt 后重启历史命令还会存在， 利用历史命令简单的方法是按 "向上键 + 回车键" 调用上一次执行的命令，以下是所有执行的历史命令调用方法：

`!`

显示历史命令的帮助信息

`!!`

再次执行上一个命令

`!:`

显示所有的历史命令

`!:n`

显示最后的 n 个历史命令

`!n`

执行index 为 n的命令，index为执行 `!:` 命令显示的index

`!-n`

执行第 n 个命令的前一个命令

`!string`

执行以 'string' 开头的最近的命令

`!?string`

执行包含 'string' 字符串的命令

配置文件 .sbt

.sbt vs .scala 构建语句定义

一个项目的构建定义可以是在项目根目录中以 `.sbt` 后缀结尾的文件，也可以是一个在子目录 `project` 下以 `.scala` 结尾的文件

这章主要讨论 `.sbt` 文件定义，这种定义已经适合大部分情况. `.scala` 定义方式典型的用在多个 `.sbt` 文件分享共用的定义语句或者是复杂的项目构建中。更多信息参考.scala 定义

什么是构建语句?

通过验证和解析构建语句文件， sbt 以一个不可变 Map（key-value 键值对）来描述构建过程结束

例如，一个 key 为 `name` 并且它的 Map 值是字符串，这个配置项 `name` 代表这个项目的名称

构建语句定义不会直接修改影响 **sbt** 的**Map**

相反，构建语句是一个由 `Setting[T]` 类型的对象构成大的列表构成，T是 Setting Map 值的类型， Setting可以通过以下几种方式修改map:添加一个新的键值对、修改以存在的key的值(在函数式编程中一个不可变数据结构和值，通过新赋值的方式来修改值，而不是在原值的基础上修改)

在build.sbt 中你可能要创建一个 `Setting[String]` 类型的配置项申明项目名称：

```
name := "hello"
```

这个 `Setting[String]` 通过添加或替换原有key值来修改，并且赋值为"hello" ，这个修改的 map 变成一个新的map。 创建一个 map，sbt 首先配置列表所有修改的配置放在一块，并且如果有的配置值依赖某些配置项将在依赖的配置项后处理。然后 sbt 利用排序后的配置项构建新的 map

总结：构建语句是一个 `Setting[T]` 组成的列表， Setting[T]是一个可以通过修改的map key-value 键值对构成， T是键值对值的类型

如何定义 build.sbt 配置项

build.sbt 是一个 Seq[Setting[_]]， 是一系列用空行分割 Scala 表达式，每行是这个序列的一个元素.

例如：

```
name := "hello"

version := "1.0"

scalaVersion := "2.10.3"
```

每个配置项都是一个 scala 表达式，在 build.sbt 中的表达式都是独立而不是一个 scala 代码块. 这些表达式由 val、lazy val 和 def 构成，对象和类不允许定义在 build.sbt 中，应该定义在子目录 `project` 中的 `.scala` 源代码文件中。

配置表达式的左值如 `name`, `version` 和 `scalaVersion` 是配置项的 key, key 是 `SettingKey[T]`, `TaskKey[T]` 或 `InputKey[T]` 的实例，其中 T 是期望值的类型，具体key的类型将在下面介绍。

Key 对象有一个方法 `:=` 调用将返回 `Setting[T]`, 你可以用 Java 语法风格调用：

```
name.:=("hello")
```

在 Scala 中允许 `name := "hello"` 方式调用（在Scala中对于单个参数的方法允许这种形式调用）

name 的 `:=` 方法返回一个 Setting 对象，其中具体的类型为Setting[String], 泛型类型 String 在 name key 本身定义中也出现了，但是 name 类型为 SettingKey[String]. 在这个例子中，将返回的 Setting[String] 对象通过添加或者替换得到一个新的 sbt 配置项 map， 并给定值为"hello"

如果给定一个错误类型的配置值，将无法编译通过

```
name := 42 // will not compile
```

配置项之间必须用空行分割

不允许在budil.sbt将配置写成如下格式：

```
// will Not compile, not blank lines
name := "hello"
version := "1.0"
scalaVersion := "2.10.3"
```

sbt 需要一个分隔符用来判断一个表达式结束和另一个表达式起始， `.sbt` 文件中包含一系列 Scala 表达式，不是单个一个 Scala 项目，这些表达式必须分隔开并且单独进行编译.

Keys

类型

- SettingKey[T]: 这类key 的值只计算一次(当加载项目的时候计算完成后将一直保留)
- TaskKey[T]: 这类 key 的值将被作为任务调用，可以被重复调用计算的，也可能会产生影响
- InputKey[T]: 这类key 是针对一个任务需要传递一些输入参数

内建 Keys

内建 Keys 是调用对象`Keys`的成员变量，对于 build.sbt 隐式包含 `import sbt.Keys._`, 所以`sbt.Keys.name`可以写作 `name`

自定义 Keys

自定义 keys 可以分别用 `settingKey`, `taskKey`和`inputKey`方法创建. 每个方法定义期望关联值的类型和该key的描述信息， 每个key 的名称保存在 val 的常量中，例如， 定义一个名为 `hello` 的任务类型的key

```
lazy val hello = taskKey[Unit]("an example task")
```

.sbt 可以包含 vals 和 defs 的定义，所有这类型的定义将在解析配置项前执行， vals 和 defs 定义必须和所有配置项配置用空行分割

注意： 一般情况下推荐用lazy val 替换用 val 可以避免初始化值得顺序问题

任务**Keys** 和 配置 **Keys**

TaskKey[T] 被称为一个任务，任务操作如 compile 或 package, 它们可能返回一个 Unit 类型 (Unit 在 Scala 中相当于 void)或返回一个关联的任务，例如 package 这个任务将返回一个TaskKey[File] 创建jar包的任务

当启动执行一个任务，例如执行 `compile` 在交互模式下， sbt 将执行和该任务相关的任务，sbt 项目描述表（map）中可以保存一个字符串的配置项（例如name配置项），也可以是保存可执行的代码块的任务（例如compile任务）， 即使一个可执行任务返回一个字符串，它也是在任何时候可重复执行的

定义 **task** 和 **settings**

可以用 `:=` 给一个配置项或任务赋值，对于配置项的值将在项目加载的时候一次性计算，对于任务在执行的时候都会重新执行计算

例如：

```
// task
hello := {println("Hello!")}

// settings
name := "hello"
```

任务和配置项的类型

Setting 通过一个任务key和一个配置项key创建出来的有细微的区别， `taskKey := 42` 结果是 `Setting[Task[T]]` 然而 `settingKey := 42` 结果是 `Setting[T]`. 对于大部分情况下基本看出来区别，因为任务key依然创建一个类型为T的值在任务执行的时候

T 和 Task[T] 不同的另一个含义： 一个配置项不能依赖一个任务，因为配置项仅在项目初始化的时候计算一次.

Keys 在 sbt 的交互模式

在sbt交互模式下，你可以执行任何任务key, 当输入compile的时候将执行 任务key为compile的任务，如果给定的key类型不是任务而是一个配置项时将输出配置项的值，如果key为任务类型的执行结果将不会输出到终端，要看任务的输出结果需要执行 `show <task name>` 而不是 `<task name>` 。习惯性的定义 sbt 的key的时候使用和Scala命名风格一样的驼峰命名。

了解一个key的更多信息，可以在交互模式下使用 `inspect <keyname>` ， 可以看到该key的值的类型和摘要描述信息等

在 build.sbt 中导入包

你可以再 build.sbt 顶部使用 `import` 语句， 它们不需要用空行分割。sbt 默认情况下隐式的导入了以下包：

```
import sbt._
import Process._
import Keys._
```

添加依赖包

要添加一个第三方库的依赖有两种方法，一种是直接将jar包放到 lib目录下，另一种方法是在build.sbt中添加依赖配置，如以下的方法：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

以上这个配置将在项目中添加一个 Apache Derby的库，并且版本为 10.4.1.3

libraryDependencies key 包含两个方法： `+=` (不是 `:=`)和 `%` ， **+=** 是在原来值上追加新值而不是替换原值，更多的解释可参考配置配置项. `%` 方法作用是构建一个 Ivy 模块ID在依赖库中被解析。

作用域

深入 Keys

前面我们一直简单的认为类似于 `name` 的 key 将一一对应一个值，在 sbt 中其实就是一个 key-value 的map 表，事实上每个 key 除了关联一个值外还有一个上下文关系，被称为“作用域”

例如：

- 如果在一个项目构建定义中含有多个项目，那么一个 key 可能有不同的值在不同的项目中
- 对于项目代码和项目测试代码中 `compile` 这个key对应的值是不同的
- `packageOptions` （包含打包jar文件的所有参数）key 在打包 class 二进制文件和源代码文件时的值是不同的

对于 `name` 这个key不是只有一个值，值会随作用域的不同而不同，但是在同一个作用域中一个key只有一个值

以前认为 sbt 是通过处理一个配置列表生成 key-value 的一个 map表来描述整个项目构建的，现在可以把这个map认为是一个由作用域的 map表，在项目构建定义中（如build.sbt文件中定义) 每个 key 都有一个对应的作用域。

一般情况下每个 key 都有一个隐含或默认的作用域，但是如果默认的不是想要的需要显式覆盖声明

作用域的维度

每个作用域维度是一个类型，每个类型实例都可以定义自己唯一值得key, 以下是三种作用域维度：

- 项目维度
- 配置维度
- 任务维度

作用域的项目维度

如果在一个构建工程中定义多个项目，每个项目拥有自己唯一的配置，那么这时key的作用域是一个具有项目维度的作用域。

作用域项目维度可以设置为“整个工程”有效（可以称为该作用域为工程作用域），这样一个配置将作用于整个构建工程中而不是单一的一个项目，构建级别的配置经常用来当做备用，当某个项目中没有配置该配置的时候。

作用域的配置维度

一个配置维度定义一个构建类型，可能有自己的 `classpath` 、 源代码目录、打包发布等，配置维度这个概念来源于 Ivy, 由于 Sbt 的包依赖管理用的是 MavenScopes

在 sbt 中的一些配置维度：

- Compile : 定义编译项目配置 (src/main/scala)
- Test : 定义测试项目的配置(src/test/scala)
- Runtime : 定义运行一个工程时的配置

默认情况下，在编译、打包、运行是所有的key将对应关联一个配置维度，所以在不同的配置维度下运行结果可能不同。最常见如任务类型的key `run` , `compile` , `package` , 其实所有的key都会受配置维度的作用域的影响，例如 `sourceDirectories` , `scalacOptions` , `fullClasspath` 等

作用域的任务维度

配置可以影响一个任务的执行，例如， `packageSrc` 会受到配置参数 `packageOption` 的影响。为了实现这个功能在 sbt 中 `packageSrc` 可以当做配置参数 `packageOption` 的作用域

打包构建有多个任务(packageSrc, packageBin, packageDoc)可以共享和打包相关的配置参数，例如 `artifactName` 和 `packageOptions` , 但是他们的值在不同的任务维度下是不同。

全局作用域

每个作用域维度都是由一个维度类型实例构成(例如任务维度就是由一个任务实例构成)， 一个维度也可以由一个全局值构成

全局的概念正如你所理解的，一个参数配置值将被应用到所有的维度实例中，例如一个任务维度是全局的，那么这个配置将在所有任务中有效。

委托

当一个作用域中没有定义某个 key, 那么其在该作用域是没有关联的值。对于每个作用域，sbt通过搜索其他作用域的路劲作为某个key 的备选作用域，典型的例子：如果一个key在指定作用域中没有关联的值，sbt试图从其他作用域获取一个值，例如全局作用域或者工程作用域。

这个特性允许你在一个作用域中设置某个key，在其他的作用域中继承该key, 可以使用sbt的inspect命令查看某个key的搜索作用域详细信息。

作用域在运行**sbt**时相关解释

在交互模式下或命令下， sbt 将用下面的形式表示作用域：

```
{<build-uri>}<project-id>/config:intask::key
```

- `{<build-uri>}/<project-id>` 表示一个项目维度的作用域，当表示整个工程的作用域时 `<project-id>` 部分可以省略
- config 表示作用域的配置维度
- intask 表示作用域的任务维度
- key 表示配置的key

> `*` 在上述任意段中出现表示在该作用域维度下是一个全局作用域

如果key省略指定某一部分，将按以下规则推断作用域：

- 如果省略项目维度将被认为当前的项目
- 如果一个依赖配置维度的key在省略配置或任务维度时将自动探测

作用域的例子详解

- `*:fullClasspath` 指定一个全局的配置，而不是默认配置
- `doc::fullClasspath` 配置在doc任务中fullClasspath参数，项目和配置维度默认
- `{file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath` 表示在工程{file:/home/hp/checkout/hello/}中的项目default-aea33a下的配置维度为test的fullClasspath配置参数，任务维度为默认的
- `{file:/home/hp/checkout/hello/}/test:fullClasspath` 表示是一个工程级别的作用域，工程为{file:/home/hp/checkout/hello/}
- `{.}/test:fullClasspath` 表示一个工程级别的作用域，这块的 `{.}.{.}` 在Scala代码中可以写成 ThisBuild
- `{file:/home/hp/checkout/hello/}/compile:doc::fullClasspath` 表示 fullClasspath 配置参数设置所有的作用域维度

检测作用域

在 sbt 交互模式下可以使用命令 `inspect` 来检测一个配置参数的作用，例如

```
> inspect test:fullClasspath
```

命令执行返回结果如下：

```
[info] Task: scala.collection.Seq[sbt.Attributed[java.io.File]]
[info] Description:
[info]  The exported classpath, consisting of build products and unmanaged and managed
, internal and external dependencies.
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath
[info] Dependencies:
[info]  test:exportedProducts
[info]  test:dependencyClasspath
[info] Reverse dependencies:
[info]  test:runMain
[info]  test:run
[info]  test:testLoader
[info]  test:console
[info] Delegates:
[info]  test:fullClasspath
[info]  runtime:fullClasspath
[info]  compile:fullClasspath
[info]  *:fullClasspath
[info]  {.}/test:fullClasspath
[info]  {.}/runtime:fullClasspath
[info]  {.}/compile:fullClasspath
[info]  {.}/*:fullClasspath
[info]  */test:fullClasspath
[info]  */runtime:fullClasspath
[info]  */compile:fullClasspath
[info]  */*:fullClasspath
[info] Related:
[info]  compile:fullClasspath
[info]  compile:fullClasspath(for doc)
[info]  test:fullClasspath(for doc)
[info]  runtime:fullClasspath
```

在第一行中可以看到一个任务key, 其值得类型是 `scala.collection.Seq[sbt.Attributed[java.io.File]].`

"Provided by" 表示该配置参数定义的作用域:

`{file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath` (fullClasspath 配置在作用域任务维度为test，作用域项目维度为{file:/home/hp/checkout/hello/}default-aea33a的作用域中)

"Dependencies": 配置参数章节解释

"Delegates": 表示如果某个key没有定义，将按照以下路劲搜索：

- 两个配置作用域（`runtime:fullClasspath` , `compile:fullClasspath`），在这些作用域中的key，项目维度没有指定默认是当前项目，任务维度没有指定默认是任务全局作用域
- 配置维度为全局的作用域（`*:fullClasspath`），项目维度没有指定默认是当前项目，任务维度没有指定默认是任务全局作用域
- 项目维度设置 `{.}` 或者ThisBuild (表示工程级别的作用域，没有指定项目)
- 项目维度设置为全局作用域(`*/test:fullClasspath`)(注意：当没有指定项目的时候表示当前项目，这块代表的意思是全局作用域，比如 `*/test:fullClasspath` 和 `test:fullClasspath` 代表的意义不一样)
- 项目和配置维度都为全局的作用域 (`*/*:fullClasspath`), 任务维度没有指定，所以当设定为 `*/*:fullClasspath` 作用域时，在作用域的三个维度上都为全局的

运行 `inspect fullClasspath` （对比上一个例子`inspect test:fullClasspath`） 会发现返回的结果有所不同，这是因为当不指定配置维度的作用域时，sbt将`inspect fullClasspath`自动探测为 `compile.inspect compile:fullClasspath`执行.

如何在工程构建中定义作用域

如果单独在build.sbt中创建一个key，这个key作用域的项目维度将为当前项目，配置和任务维度将为全局作用域:

```
name := "hello"
```

运行命令`inspect name`将看到"Provided by"为：`{file:/home/hp/checkout/hello/}default-aea33a/*:name`， 表示作用域项目维度为 `{file:/home/hp/checkout/hello/}default-aea33a`, 作用域任务维度为`*`(全局)， 作用域任务维度没有指定默认代表全局, build.sbt 定义是针对单个项目的，所以“当前项目”指的就是当前build.sbt定义的项目(对于多项目构建,每个项目有对应一个build.sbt)

所有的key都有一个方法用来设定作用域，其参数可以为作用域的任何一个维度的对象实例。例如，可以通过如下方式将`name`配置设置为配置维度为`Compile` 的作用域：

```
name in Compile := "hello"
```

或者可以将`name`配置设置为任务维度为`packageBin`的作用域（当然这例子有点不合适）

```
name in packageBin := "hello"
```

当然了也可以为一个key指定多个作用域维度,例如将key `name` 同时指定配置维度和任务维度：

```
name in (Compile, packageBin) := "hello"
```

也可以指定一个key的作用域为全局：

```
name in Global := "hello"
```

(`name in Global` 隐式将作用域转化为一个对于所有维度都为全局的作用域，默认情况下任务和配置维度的作用域已经是全局的了，但是这块会影响项目维度的作用，因为其隐式转化为`*/*:name`而不是`{file:/home/hp/checkout/hello/}default-aea33a/*:name`)

如果没有用过Scala语言，理解 `in` 和 `:=` 这两个方法很重要，推荐用scala语法形式配置，但是也可以用Java语法形式配置：

```
name.in(Compile).:=("hello")
```

什么时候指定作用域

当定义一个key在默认作用域下会有问题时需要指定作用域，例如`compile`任务类型的配置，默认作用域是在 `Compile` 或`Test`配置维度下，不会存在于其他作用域中。

如果修改`compile`这个任务配置的值需要指定其作用域，修改语句如`compile in Compile`或`compile in Test`, 如果单独写作为`compile` sbt 会重新创建一个作用域为当前项目的任务配置，而不是去修改在配置维度作用域下的标准compile任务配置。

如果得到一个错误信息“Reference to undefined setting”， 一般情况下是因为配置项指定作用域失败或者指定了一个错误的作用域。当定义的key可能在其他作用域中已经定义会接收到“Did you mean compile:compile?”错误提示信息

一般会简单的认为配置项就是一个key-value键值对，其实对于所有的配置项都会包含一个key-value 和一个对应的作用域（作用域有三个维度），如配置表达式：`packageOptions in (Compile, packageBin)`, 当配置为`packageOptions`也是一个合法的配置项，只是该配置项的作用域将是默认的作用域(项目维度为当前项目，任务和配置维度为全局)

配置参数的方法

回顾：配置项

一个工程构建定义一个 `Setting` 类型的列表，通过sbt转化为sbt的描述数据结构（key-value键值对），Setting作为一个转化前的输入类型，转化后输出一个 map表。

不同的配置项有不同的转化方法，例如前面的 `:=` 方法。 一个 Setting类型的配置项可以通过 `:=` 转化成一个值为一个常量的map表，例如，转化一个配置项 `name := "hello"` 是将hello 赋值给该配置项的key

Settings must end up in the master list of settings to do any good (all lines in a build.sbt automatically end up in the list, but in a .scala file you can get it wrong by creating a Setting without putting it where sbt will find it).

配置值追加操作： **+=** 和 **++=**

通过直接赋值方法 `:=` 是最简单的一种转化方法， 所有的配置项还有其他的方法，在 `SettingKey[T]` 中泛型 T 如果是一个序列类型的话，支持追加操作不仅仅是重新赋值替换操作。

- `+=` 操作是追加单个元素到序列中
- `++=` 操作是追加一个序列到序列中

比如, 对于配置 `sourceDirectories in Compile` 值得类型是 `Seq[File]`, 默认情况下这个配置已经有包含了目录 `src/main/scala` 。如果还想编译目录 `source` 下的源代码可以添加该目录：

```
sourceDirectories in Compile += new File("source")
```

或者，利用 sbt 包中提供的函数 `file()` 更加方便

```
sourceDirectories in Compile += file("source")
```

file() 函数也是创建 File 对象

也可以用操作符 `++=` 一次性添加多个目录：

```
sourceDirectories in Compile ++= Seq(file("sources1"), file("sources2"))
```

其中Seq(a, b, c, ...)是 Scala的标准语法，用来创建一个序列

当然了除了追加操作还支持直接赋值替换配置原值：

```
sourceDirectories in Compile := Seq(file("sources1"), file("sources2"))
```

通过其他配置项计算一个配置项

引用其他的任务配置或参数配置的值通过调用任务或参数配置，通过方法 `:=` 、 `+=` 或 `++=` 来引用一个配置

比如，定义一个项目组织和项目名称一样的配置：

```
// organization 接收的值类型和 name一样都是 Setting[String]
organization := name.value
```

或者通过引用项目目录来定义项目名称：

```
// name 是Key[String]类型，baseDirectory是Key[File]
name := baseDirectory.value.getName
```

由于baseDirectory的值类型和name的值类型不一样，需要调用java的java.io.File类库 getName 方法转化

sbt同时支持引用多个配置项的值，例如：

```
name := "project " + name.value + " from " + organization.value + " version " + versio
n.value
```

参数配置依赖

在配置 `name := baseDirectory.value.getName` 中，参数配置 name 依赖配置baseDirectory，如果在放有上述配置的build.sbt的目录下交互模式运行 `inspect name` 命令，将返回如下的提示信息：

```
[info] Dependencies:
[info]  *:baseDirectory
```

sbt 是很容易的探测出各个参数配置间的依赖关系的，包括任务配置也可以依赖参数配置，或者任务配置间相互依赖， 例如运行 `inspect compile` 会发现 compile 任务配置会依赖参数配置compileInputs， `inspect compileInputs` 还会发现参数配置compileInputs又依赖其他的参数配置，等等。最后会发现形成一个依赖链，这些依赖关系是在sbt编译的时候自动探测计算的。所以所有的依赖关系都是自动计算的，不需要显示的声明。

未定义配置

当用 `:=` 、 `+=` 或 `++=` 方法去引用一个配置，所依赖的配置必须存在，否则sbt会报错误信息"Reference to undefined setting", 如果报错需要确认依赖的配置是否在指定作用域中存在。

也可能会出现循环依赖的配置错误，这个时候sbt会提示错误

任务配置引用参数配置

一个任务配置可以依赖参数配置或其他的任务配置，通过 `def.task` 或操作符 (`:=` 、 `+=` 或 `++=`)引用配置

比如，一个源代码生成任务将依赖项目根目录和编译classpath两个参数配置：

```
sourceGenerators in Compile += Def.task {
  myGenerator(baseDirectory.value, (managedClasspath in Compile).value)
}.taskValue
```

任务配置依赖

如[配置文件 .sbt]所述，一个任务配置用赋值方法 `:=` 操作的参数是 `Setting[Task[T]]` 而不是 `Setting[T]` 等， 任务配置输入参数可以接收一个参数配置作为输入，但是一个参数配置是不可以接收一个任务配置作为输入

以下为两个配置：

```
val scalacOptions = taskKey[Seq[String]]("Options for the Scala compiler.")
val checksums = settingKey[Seq[String]]("The list of checksums to generate and to veri
fy for dependencies.")
```

（scalacOptions和checksums之间没有任何联系，它们只是配置值得类型相同，但是其中scalacOptions是一个任务配置）

在build.sbt中允许一个任务配置依赖参数配置，比如:

```
// 合法的配置
scalacOptions := checksums.value
```

是合法的配置方法，但是如果将一个参数配置配置成依赖一个任务配置的时候就会报错，因为参数配置在项目加载的时候只计算一次，可以当做常量处理，但是任务配置是在不断的重复的运行的。

```
// 非法的配置
checksums := scalacOptions.value
```

追加依赖操作：**+=** 和 **++=**

一些配置可以被追加引用到一个已经存在的配置中，和直接赋值操作 `:=` 一样.

例如，有一个项目覆盖率报告文件（文件名引用项目名称的参数配置）需要删除，可以通过如下配置：

```
cleanFiles += file("coverage-report-" + name.value + ".txt")
```

Lib库依赖

添加Lib库依赖关系有两种方式：

- 非管理依赖方式，是通过将依赖的Jar包放到项目的lib目录
- 管理依赖方式，是在工程构建配置中配置依赖关系，sbt会自动从托管代码库中下载依赖库

非管理依赖方式

很多人用管理依赖的方式替代非管理方式，其实非管理方式用起来非常方便。非管理依赖方式的工作原理就是将jar包放到lib目录下，sbt会自动的将其添加到classpath中。也可以将一些测试依赖放到lib目录下，如ScalaCheck、Specs2和ScalaTest这些依赖包

lib目录下载的依赖在所有的classpath中有效（对于编译、测试、运行或命令终端），如果想修改某个配置维度的作用域的classpath配置，需要按照如下修改方式 `dependencyClasspath in Compile` 或 `dependencyClasspath in Runtime`

对于非管理方式的依赖不需要在build.sbt额外配置，但是如果想自定义一个依赖目录而不是默认的lib目录时，可以通过修改 `unmanagedBase` 参数配置, 比如用目录 `custom_lib` 替换 `lib` 目录

```
unmanagedBase := baseDirectory.value / "custom_lib"
```

其中 `baseDirectory` 是项目的根目录，修改 `unmanagedBase` 参数配置依赖 `baseDirectory` ,关于参数配置的依赖可以参考配置参数的方法

在非管理依赖方式中还提供了一个任务配置 `unmanagedJars` 用来列举 `unmanagedBase` 所有的jar包的，在多项目构建中或更加复杂的构建中可能需要修改 `unmanagedJars` 配置来完成。例如，在Compile这个配置维度作用域下要清空所有非管理方式的jar包，可以利用如下配置：

```
unmanagedJars in Compile := Seq.empty[sbt.Attributed[java.io.File]]
```

管理依赖方式

sbt 利用Apache Ivy方式管理依赖包，如果熟悉 `Ivy` 或 `Maven` 的话，理解sbt的管理依赖包将会非常的容易。

`libraryDependencies` 参数配置

一般情况下只需要通过配置 `libraryDependencies` 参数配置即可设置依赖的包，也可以通过编写 `Maven` 的POM配置文件或 `Ivy` 的配置文件来扩展包依赖的功能。

以下是申明一个包依赖关系， 其中`groupId`，`artifactId`和`revision`是字符串类型

```
libraryDependencies += groupID % artifactID % revision
```

或用如下申明，其中`configuration`是一个字符串或者一个配置维度实例

```
libraryDependencies += groupID % artifactID % revision % configuration
```

libraryDependencies这个配置key在Keys中声明语句如下

```
val libraryDependencies = settingKey[Seq[ModuleID]]("Declares managed dependencies.")
```

从上述 libraryDependencies key 申明语句中可以看出其值是接收一个由ModuleID对象构成的序列。在申明依赖关系的语句中有`%`方法，该方法是用字符串创建一个ModuleID类型的对象。

当然sbt必须知道所配置的依赖库到什么地方下载，如果配置的依赖库在默认的远程库中存在将直接下载，比如`Apache Derby`就是在标准远程库`Maven2`中：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

如果配置到build.sbt 并且执行`update`， sbt将自动将其下载到`~/.ivy2/cache/org.apache.derby/`目录下(由于`compile`任务配置会依赖`update`这个任务，所以一般情况不用手动执行update)

当然，也可以用 `++=` 方法一次性添加一个依赖列表:

```
libraryDependencies ++= Seq(
  groupID % artifactID % revision,
  groupID % otherID % otherRevision
)
```

在少数情况下可能也会用到`:=`赋值方法

指定依赖库Scala版本

如果用 `groupID %% artifactID % revision`而不是`groupID % artifactID % revision` 方式配置依赖关系(二者区别在于前者的groupId后的是两个`%`)， sbt将会添加当前scala版本到依赖的报名后，这个只是一个快捷的方式，你也可以直接硬编码scala版本：

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.11.1" % "0.3"
```

假如当前构建项目的`scalaVersion`为2.11.1,如下方式和上述的结果是一样(利用`%%`方式添加依赖库的Scala版本)

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

在多个Scala版本下构建项目，可以使用这种方法来匹配二进制兼容的对应依赖包。

复杂的情况是经常有依赖包对于不同的Scala版本间有细微的差别，所以如果依赖包在`2.10.1`版本下存在，但是当前项目scalaVersion := "2.10.4", 将会发现用`%%`是获取不到依赖包的，这时需要确定该版本的依赖包是否可用，并且可以通过硬编码版本的方式添加依赖

Ivy 自动匹配版本

在配置`groupID % artifactID % revision`中`revision`不是一定要指定一个固定的版本号，Ivy可以自动选择一个高版本包根据指定的限定条件，例如如果要替换一个固定版本"1.6.1"，可以通过指定"latest.integration","2.9.+"或"[1.0,)"来替换，更多的配置请参看Ivy配置文档

远程依赖库地址管理

不是所有的包在同一个远程库中都存在，sbt默认用的是标准的`Maven2`远程库，如果依赖的包不在默认的远程库中，需要手动指定一个远程库供Ivy查找。

添加远程库的方法如下：

```
resolvers += name at location
```

例如：

```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositorie
s/snapshots"
```

`resolvers` 这个配置项在 `Keys` 中的定义如下：

```
val resolvers = settingKey[Seq[Resolver]]("The user-defined additional resolvers for a
utomatically managed dependencies.")
```

在添加远程库配置语句中的 `at` 方法是将字符串转化为`Resolver`对象。

sbt 也支持搜索本地的`Maven`依赖库，配置如下：

```
resolvers += "Local Maven Repository" at "file://"+Path.userHome.absolutePath+"/.m2/re
pository"
```

也可以简写为：

```
resolvers += Resolver.mavenLocal
```

重写默认的远程依赖库

参数配置 `resolvers` 中是不包含默认的远程库配置的，仅仅用来配置添加远程库地址，sbt 最终是通过合并 `resolvers` 配置和 `externalResolvers` 配置来得到远程库地址集合，所以如果要修改默认远程库的话需要修改参数配置 `externalResolvers`

配置维度的作用域依赖库

经常在测试代码中会用到依赖库（在src/test/scala目录中的代码将通过配置维度为Test作用域的配置来编译）但在项目代码中不会用到。

如果只想在编译测试代码的时候加入到classpath而在编译项目代码的时候不加入，可以通过添加% "test"指定配置维度的作用域来限定：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % "test"
```

也可以指定配置维度的一个实例对象来限定：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % Test
```

那么，当限定到某个配置维度作用域时利用命令 `show compile:dependencyClasspath` 查看配置时将不会看到 `derby` jar包，但是如果查看 `show test:dependencyClasspath` 将会看到 `derby` jar包在列表中。

常见的，测试相关的包ScalaCheck、Specs2和ScalaTest将指定% "test"来限定在Test配置维度下使用。

多项目构建

多项目

通常在一个工程中构建多个项目间会有关联，尤其是它们都依赖一个项目时可以很容易的更新项目

在一个工程中每个子项目都会有自己的源代码目录、生成各自的jar包当执行 `package` 时.

一个项目通过申明一个 `Project` 类型的懒值来定义，例如：

```
lazy val util = project

lazy val core = project
```

这个变量值名称将被用来当做 `Project Id` 和项目的根目录名称，这个ID将用来在命令行中引用项目，利用方法 `in` 可以修改默认的项目根目录。例如， 以下是更加明确的申明项目：

```
lazy val util = project.in(file("util"))

lazy val core = project in file("core")
```

项目间依赖

在一个工程中一个项目完全可能依赖另一个项目，经常有两种依赖方式：聚合和classpath

项目聚合

聚合的意思是当运行一个任务在一个项目中，其通过聚合方式依赖的项目也会执行，例如：

```
lazy val root = (project in file(".")).
  aggregate(util, core)

lazy val util = project

lazy val core = project
```

在上面的例子中， `root` 项目聚合了项目 `util` 和 `core` ，编译 `root` 项目将看到三个项目被编译。

在项目聚合过程中，以 `root` 项目为例，是可以控制任务维度作用域的配置的，例如,可以控制在执行 `update` 这个任务时不进行聚合：

```
lazy val root = (project in file(".")).
  aggregate(util, core).
  settings(
    aggregate in update := false
  )

[...]
```

`aggregate in update` 表示在 `update` 这个任务维度作用域中aggregate的配置。（具体可参考作用域章节）

> 注意：在聚合过程中聚合是并行处理的，所以被聚合的项目时没有先后顺序的

classpath 依赖

在源代码层级一个项目可能会依赖另一个项目，可以通过 `dependsOn` 方法添加依赖关系，例如， `core` 项目需要在classpath中指定 `util` 项目，可以这样定义 `core` 项目:

```
lazy val core = project.dependsOn(util)
```

这样就可以在 `core` 项目中调用 `util` 项目的方法， 当编译的时候会有编译顺序， `util` 必须在 `core` 之前编译， 如果依赖多个项目，可以给 `dependsOn` 方法指定多个参数，例如， `dependsOn(bar, baz)`

配置维度作用域的 classpath 依赖

`foo dependsOn(bar)` 表示foo在 `Compile` 这个配置维度作用域下依赖在配置维度作用域为 `Compile` 下的 `bar` 项目，确切的写法应该为： `dependsOn(bar % "compile->compile")` ，在"compile->compile"中的 `->` 表示项目间的依赖关系，所以"test->compile"可以表示为在 `Test` 配置维度作用域下的foo依赖 `Compile` 配置维度作用域下的bar

省略"->config"这部分隐含意思为"->compile", 所以 `dependsOn(bar % "test")` 意思是在 `Test` 配置维度作用域下的foo依赖 `Compile` 配置维度作用域下的bar

可以声明为"test->test"，意思为在 `Test` 配置维度作用域下项目依赖，例如，在 `src/test/scala` 目录的公共类库可以在 `src/test/scala` 源代码中使用。

针对一个依赖关系可以配置多个配置维度的作用域，用分号分隔，例如： `dependsOn(bar % "test->test;compile->compile")`

默认根项目

如果一个项目没有在工程根目录下定义，sbt将创建一个聚合整个工程子项目的默认项目。

由于项目hello-foo 定义了 `base = file("foo")` ,项目目录为子目录foo， 源代码可以直接放到foo目录中，类似 `foo/Foo.scala` 或放到 `src/main/scala` ,如果是采用sbt工程构建标准目录，foo目录下还包括工程构建定义文件。

在foo目录下的任何 `.sbt` 文件将被合并在一起，并且定义的配置项属于项目维度 `hello-foo` 作用域。

在hello工程中允许子项目配置不同的版本，可以在配置文件: `hello/build.sbt, hello/foo/build.sbt, 和hello/bar/build.sbt` 中配置不同版本，现在在交互模式下执行 `show version` 将看到如下信息：

```
> show version
[info] hello-foo/*:version
[info]  0.7
[info] hello-bar/*:version
[info]  0.9
[info] hello/*:version
[info]  0.5
```

`hello-foo/*:version` 被定义在 `hello/foo/build.sbt` 中 `hello-bar/*:version` 被定义在 `hello/bar/build.sbt` 中 `hello/*:version` 被定义在 `hello/build.sbt` 中

每个version是不同的项目维度作用域，但是这三个build.sbt部分配置是相同的

> 每个项目的配置都在自己项目目录下的 .sbt 文件中配置，其实对于上述配置有更简单的方法，那就是配置到 .scala 文件中，列举出项目和对应的项目根目录

你会发现将多个项目按顺序定义在 .scala 配置文件中会比单独定义在各自的目录下更加清晰，不过这个自己决定。

不允许定义一个子目录为 `project` 的目录， `foo/project/Build.scala` 将会被忽略

交互模式下操作项目

在sbt的交互模式下，使用命令 `projects` 列举出该工程的所有项目，命令 `project <projectname>` 可以切换当前项目。当运行一个任务的时候（比如 `compile` ）将在当前项目下运行，所以没有必要在根项目中编译，可以单独编译一个子项目。

配置代码复用

.sbt 配置文件中的定义在多个 .sbt 中是相互不可见的，为了是多个.sbt间复用配置，可以在根目录的project目录下定义一个或多个 .scala 的文件，这个目录其实也是一个sbt工程，只不过这个的作用是工程构建。

例如：

`<root>/project/Common.scala:`

```
import sbt._
import Keys._

object Common {
  def text = "org.example"
}
```

在.sbt中可以直接调用：

`<root>/build.sbt:`

```
organization := Common.text
```

插件使用

什么是插件？

插件可以扩展工程构建定义，通常是增加一些新的配置，配置可以是任务配置，例如，一个插件可以增加一个 `codeCoverage` 的任务配置，用来生成项目单元测试的代码覆盖率报告。

使用插件

如果一个项目的目录为hello, 并且在该项目中使用 `sbt-site` 这个插件，只需创建一个名为 `hello/project/site.sbt` 配置文件，并且通过 `addSbtPlugin` 方法申明该插件的Ivy模块ID：

```
addSbtPlugin("com.typesafe.sbt" % "sbt-site" % "0.7.0")
```

如果添加一个 `sbt-assembly` 的插件，创建一个 `hello/project/assembly.sbt` 配置文件，并且配置如下：

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

并不是每个插件在默认的远程库中都存在，这个需要根据具体的插件文档来配置远程库地址：

```
resolvers += Resolver.sonatypeRepo("public")
```

插件一般都会提供一些配置来添加启用插件，下一小节讨论这个问题。

插件的启用

一个插件通过配置可以自动的添加到项目中，不需要额外的做任何工作，在0.13.5有一个新的特性是可以自动启用和配置在一个项目中，大部分自动插件都已经自动的进行了默认的配置，不过有个别的插件也需要明确启用。

如果一个插件需要配置启用才可以使用，需要在build.sbt中进行如下配置：

```
lazy val util = (project in file("util")).
  enablePlugins(FooPlugin, BarPlugin).
  settings(
    name := "hello-util"
  )
```

`enablePlugins` 方法用来指定需要启用的配置。

一个项目也可以通过 `disablePlugins` 来排除使用一个插件。例如，在util项目中不想用插件 `IvyPlugin` ，可以如下配置：

```
lazy val util = (project in file("util")).
  enablePlugins(FooPlugin, BarPlugin).
  disablePlugins(plugins.IvyPlugin).
  settings(
    name := "hello-util"
  )
```

自动插件应该在文档中明确指定该插件是否需要显式的启用， 如果疑惑一个插件是否已经启用，可以在交互模式下使用命令 `plugins` 来判断,例如：

```
> plugins
In file:/home/jsuereth/projects/sbt/test-ivy-issues/
        sbt.plugins.IvyPlugin: enabled in scala-sbt-org
        sbt.plugins.JvmPlugin: enabled in scala-sbt-org
        sbt.plugins.CorePlugin: enabled in scala-sbt-org
        sbt.plugins.JUnitXmlReportPlugin: enabled in scala-sbt-org
```

这块显示的是所有默认开启的插件，sbt 默认启用三个插件：

- CorePlugin: 提供并行任务的控制
- IvyPlugin： 提供依赖模块的发布、解析机制
- JvmPlugin： 提供编译、测试、打包 Scala/Java代码机制

另外 `JUnitXmlReportPlugin` 插件只要是提供生成 junit-xml 文件的支持

添加老版非自动插件经常需要显式的配置，插件文档将会指出如何进行配置，典型的老版插件是由基础配置和自定义配置构成。

例如，对于 `sbt-site` 这个插件，用如下配置来启用该插件：

```
site.settings
```

如果是多项目构建，可以直接在项目配置中配置：

```
// don't use the site plugin for the `util` project
lazy val util = (project in file("util"))

// enable the site plugin for the `core` project
lazy val core = (project in file("core")).
  settings(site.settings : _*)
```

全局插件

插件可以通过在 `~/.sbt/0.13/plugins/` 下一次申明配置应用在多个项目中， `~/.sbt/0.13/plugins/` 是一个sbt项目，其classpath将导出到每个sbt构建项目中，粗略的讲，在 `~/.sbt/0.13/plugins/` 中的任何 `.sbt` 或 `.scala` 配置都会影响到所有的sbt构建项目

可以在 `~/.sbt/0.13/plugins/build.sbt` 通过 `addSbtPlugin` 方法添加一个插件来应用到所有的项目中，所以如果添加一些针对机器的环境变量等的场合这个特性非常实用。

已经存在的插件

已经存在的插件列表

一般比较常用的有两类：

- 针对某些IDE的插件
- 针对Web框架支持的插件，如xsbt-web-plugin

自定义配置和任务

定义一个配置

在配置文件 .sbt这章已经将了如何定义个配置，大部分配置定义在Default中

配置有三种类型，其中SettingKey和TaskKey已经在配置文件 .sbt介绍了，InputKey在任务配置的输入章节介绍。

对于配置的一些例子：

```
val scalaVersion = settingKey[String]("The version of Scala used for building.")
val clean = taskKey[Unit]("Deletes files produced by the build, such as generated sour
ces, compiled classes, and task caches.")
```

创建配置的构造方法有两个参数，分别是配置的名称("scalaVersion")和一个描述该配置的字符串("The version of scala used for building.")。

在配置文件 .sbt中介绍在SettingKey[T]中的T表示该配置值得类型，在TaskKey[T]中T代表该任务返回的类型。并且也介绍了一个参数配置是一个常量，其在项目加载的时候就初始化好了，而一个任务配置是可以重复执行的（任何时候在交互模式下或批量脚本中都可以调用执行）

所有的配置都定义在 `.sbt` , `.scala` 文件或插件中，任何的在 `.scala` 配置文件中或在插件中的配置都将自动的合并到 `.sbt` 文件中。

实现一个任务

当定义完一个任务配置后，需要实现一个任务配置。可以自己实现一个任务，也可以重载一个已经存在的任务，两者配置的方式没有任何区别。通过 `:=` 方法来关联任务配置来实现一个任务，例如：

```
val sampleStringTask = taskKey[String]("A sample string task.")

val sampleIntTask = taskKey[Int]("A sample int task.")

sampleStringTask := System.getProperty("user.home")

sampleIntTask := {
  val sum = 1 + 2
  println("sum: " + sum)
  sum
}
```

如果一个任务有依赖关系，可以直接引用依赖的配置，在配置参数的方法 一章已经介绍了。

实现任务最难的部分是用Scala代码实现该任务具体执行过程，例如，可以编写一个格式化HTML的任务用相关的HTML lib 库（自己可以定义添加一个依赖库，在该基础上编写)

sbt 也提供了一些工具库，例如常用的文件和目录操作API等。

利用插件

如果有许多通用的代码，可以将其提取到一个插件中，可以实现多项目的复用。

配置文件 .scala

sbt的递归性

build.sbt 是非常简单的，其隐藏了sbt真正工作的一些细节，sbt 是由Scala语言编写的，其自身也需要构建，那么由什么好的办法来实现呢？

`project` 目录是在构建项目中的另一个项目，它负责整个项目的构建定义，理论上在 `project` 目录下还可以有另一个 `project` 项目(递归)，其构建的是sbt项目本身用来支撑上级项目的构建。

例如，你可以在构建项目下再次创建一个项目，以下是目录层级结构：

```
hello/                  # your project's base directory

    Hello.scala         # a source file in your project (could be in
                        #   src/main/scala too)

    build.sbt           # build.sbt is part of the source code for the
                        #   build definition project inside project/

    project/            # base directory of the build definition project

        Build.scala     # a source file in the project/ project,
                        #   that is, a source file in the build definition

        build.sbt       # this is part of a build definition for a project
                        #   in project/project ; build definition's build
                        #   definition


        project/        # base directory of the build definition project
                        #   for the build definition

            Build.scala # source file in the project/project/ project
```

不用担心，大部分情况下是不需要创建这个的，但是理解这个概念对运用sbt很有帮助。

顺便说一下，任何以 `.sbt` 或 `.scala` 后缀的定义文件都会被用到，经常说的build.sbt或Build.scala命名只是为了方便而已，也就是说sbt配置支持多文件配置。

.scala 配置文件定义

.sbt文件定义将被合并到子目录 `project` 中，例如如下项目目录结构：

```
hello/                  # your project's base directory

    build.sbt           # build.sbt is part of the source code for the
                        #   build definition project inside project/

    project/            # base directory of the build definition project

        Build.scala     # a source file in the project/ project,
                        #   that is, a source file in the build definition
```

在build.sbt中的 Scala 配置表达式将会被编译合并到Build.scala中（或者是在project目录下的任何 `.scala` 文件中）。 在项目根目录的 `*.sbt` 文件将会变成在根目录下 `project` 构建项目定义的一部分。 `.sbt` 只是为了定义项目方便。

build.sbt 和 Build.scala 的关系

对于在构建项目定义中混合的 `.sbt` 和 `.scala` 定义，你需要理解它们之间的关系，以下是两个文件的例子，首先，假设一个项目 hello， 创建 `hello/project/Build.scala` 如下：

```
import sbt._
import Keys._

object HelloBuild extends Build {
  val sampleKeyA = settingKey[String]("demo key A")
  val sampleKeyB = settingKey[String]("demo key B")
  val sampleKeyC = settingKey[String]("demo key C")
  val sampleKeyD = settingKey[String]("demo key D")

  override lazy val settings = super.settings ++
    Seq(
      sampleKeyA := "A: in Build.settings in Build.scala",
      resolvers := Seq()
    )

  lazy val root = Project(id = "hello",
    base = file("."),
    settings = Seq(
      sampleKeyB := "B: in the root project settings in Build.scala"
    ))
}
```

创建 `hello/build.sbt` 如下：

```
sampleKeyC in ThisBuild := "C: in build.sbt scoped to ThisBuild"

sampleKeyD := "D: in build.sbt"
```

启动sbt的交互模式，输入 `inspect sampleKeyA` 将会看到：

```
[info] Setting: java.lang.String = A: in Build.settings in Build.scala
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}/*:sampleKeyA
```

然后输入 `inspect sampleKeyC` 显示如下：

```
[info] Setting: java.lang.String = C: in build.sbt scoped to ThisBuild
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}/*:sampleKeyC
```

"Provided by" 显示这两个参数配置的作用域是相同的，在 `.sbt` 文件中配置 `sampleKeyC in ThisBuild` 等同于在 `.scala` 文件中的Build.settings 中配置的值，在上述两个地方配置的值得作用域都是工程级别的作用域。

现在，在查看sampleKeyB：

```
[info] Setting: java.lang.String = B: in the root project settings in Build.scala
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}hello/*:sampleKeyB
```

sampleKeyB 的作用域是项目维度的(`{file:/home/hp/checkout/hello/}hello`)不再是工程级别的作用域。

你可能猜到 `inspect sampleKeyD` 和sampleKeyB的一样：

```
[info] Setting: java.lang.String = D: in build.sbt
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}hello/*:sampleKeyD
```

sbt 在 `.sbt` 文件中追加配置Build.settings 和 Project.settings 配置的优先级比 `.scala` 文件的高，所以配置在 `.sbt` 文件中的sampleC 或 sampleD会修改Build.scala的配置。

另一个需要注意的是：sampleKeyC 和 sampleKeyD 可以定义在build.sbt 中定义，这是因为sbt会将Build 对象自动隐式的导入到 `.sbt` 文件中，例如这个例子sbt 会将 `HelloBuild._` 隐式的导入到build.sbt 文件中。

总结：

- 在 `.scala` 文件中，可以定义 Build.settings f配置项供sbt查找，其作用域自动为工程级别的作用域
- 在 `.scala` 文件中，可以定义 Project.settings配置项供sbt查找，其作用域自动为项目维度的作用域
- 在 `.scala` 文件中的任何Build对象，都会自动的导入到所有的 `.sbt` 文件中
- `.sbt` 文件中的配置将会被追加到 `.scala` 文件中
- 配置在 `.sbt` 文件中的配置默认是项目维度的作用域，除非手动指定其他作用域

什么时候用到 .scala 配置文件

`.scala` 配置文件可以用任何的Scala代码编写，包括顶级的类和对象，由于它是标准的Scala语法，所以也没有必须添加空行来分割配置的限制

在配置参数配置推荐用 `.sbt` 配置文件，当要实现一些任务配置或者要定义一些复用配置供多个 `.sbt` 文件使用的情况推荐使用 `.scala` 配置文件

在交互模式下构建项目

你可以在交互模式下，切换到在 `project` 目录下的构建工程项目的项目中，当切换到该项目中后可以执行一些操作，如 `reload plugins.` 等

```
> reload plugins
[info] Set current project to default-a0e8e4 (in build file:/home/hp/checkout/hello/pr
oject/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/project/Build.scala)
> reload return
[info] Loading project definition from /home/hp/checkout/hello/project
[info] Set current project to hello (in build file:/home/hp/checkout/hello/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/hw.scala)
>
```

正如上面的例子，你可以用 `reload return` 命令来退出当前工程构建项目的项目，回到常规的项目中。

不可变配置

sbt 配置可能被错误的理解为在build.sbt的配置将会被添加到Build 或Project 对象的settings 字段中，其实settings 是 Build 和 Project的列表，build.sbt中的配置将会被和一个不可变的配置列表连接起来生成一个新的列表供sbt使用的， Build 和 Project中的“不可变配置”列表仅仅是完成sbt配置的一部分。

事实上，还有其他的配置文件，它们将会按照如下顺序添加：

- 配置在 `.scala` 文件中的Build.settings和Project.settings配置项
- 用户全局配置，例如在 `~/.sbt/0.13/global.sbt` 文件中可以配置影响所有工程构建的配置项
- 通过被插件注入的配置项
- 项目中 `.sbt` 文件配置的配置项
- 构建工程项目的项目(例如：所有项目中的project项目)从全局插件中添加的配置项

后面的配置将会重载前面的配置，最后生成一个配置列表。

总结

sbt核心概念

- 基于Scala，由于sbt是基于Scala编写的，所以相关的配置语法和Scala很相似
- .sbt 工程构建定义
- 可以定义一个大的 Setting对象序列，最后被转化为一个key-value键值对供sbt使用
- 可以通过 `:=` , `+=` 或 `++=` 配置一个配置
- 配置是不可变的，仅可以通过转换来修改。例如，一个 Setting对象转换成一个key-value键值对后是构建一个新的Map对象来修改的配置的，原值没有任何修改
- 每个配置都有一个类型，通过key的定义来决定
- 任务是一种特殊的配置，它是可以重复调用运算的，而参数配置是在项目加载中只初始化一次
- 作用域
- 每个配置可能有多个值在不同的作用域中
- 作用域有三个维度：项目、配置、任务
- 作用域概念使得一个配置可以再每个项目、每个任务、每个配置下都可能产生不同的行为
- 配置维度指的是构建类型，如对于主项目(main)为(Compile)或测试为(Test)
- 项目维度也支持工程级别的作用域
- 作用域具有备选和委托的特性
- .sbt vs .scala 配置文件定义
- 将参数配置放到 `.sbt` 配置文件中，将任务配置或者大的代码段配置放到 `.scala` 配置文件中
- 插件可以扩展配置
- 通过 `addSbtPlugin` 方法添加一个插件

如果以上的概念有任何疑惑可以寻求帮助、重新返回去阅读或者在交互模式下做一些试验。。

祝好运！